Latent Factor Models for Movie Recommender Systems

Chiang, Michelle - mkchiang@wisc.edu - 906 514 9420
Cuadros, Giancarlo - cuadrosmoren@wisc.edu - 906 855 6738
Wang, Qisi - qisi.wang@wisc.edu - 906 634 5944

December 16, 2016

1 Executive Summary

Life is better with recommendations. Recommender systems are simply methods of matching users to items they will like, and these systems are everywhere: Amazon.com, Pandora music, Netflix movies, Facebook friend suggestions, etc. With the emergence of the internet age, this has become a huge area of interest for researchers in optimization. Two broad classes of approaches for generating these computerized recommendation systems are content-based and collaborative filtering. This lab focuses on introducing collaborative filtering, as well as implementing and comparing two methods for model based-collaborative filtering. Specifically, these are stochastic gradient descent (SGD) and alternating least squares (ALS). This lab will utilize the MovieLens data set with over 100,000 movie ratings to train two different models that serve as recommender systems of given movies to the various users. One model will be implemented using SGD and the other with ALS. The strategies outlined here use what is called matrix factorization, which is similar to the matrix decomposition in singular value decomposition (SVD). The warm up serves to gain an appreciation for the mathematical basis of the two learning algorithms. After going through the warm up tutorial, you will have derived the fundamental equations for each of the algorithms. Additionally, you will have constructed from the available movie data a reasonable ratings matrix R that will be used in the lab. The lab itself will focus on implementing movie recommender systems using ALS, SGD, and SVD, and comparing the performance of each.

2 Recommender Systems

2.1 Introduction

Recommender systems emcompasses a variety of methods which can provide accurate predictions on items that a potential user could be interested in. Essentially, they are systems that help users discover items they may like. Big companies such as Netflix, Amazon, and Spotify incorporate these kinds of systems. They provide a scalable way of personalizing content for users and earn major revenue as a result. Back in the day, people were required to hear about recommendations on items from other people, or required to look up ratings themselves with the realization of the internet. However, the problem was that this required people to have friends that shared similar taste or have direct easy access to these ratings. With recommender systems, it is as if each user can now have their personal discovery assistant. Research into this area of data science has resulted in many approaches. Generally speaking, recommender systems are based on one of two approaches - content based and collaborative based content filtering. In this lab, we will be focusing on a branch of collaborative based content filtering called a latent factor model.

2.2 Collaborative vs. Content-based Filtering

With content-based approaches, much information about item content is needed in order to make any prediction on user preferences. In other words, the predictions heavily rely on available information regarding various item features (i.e. movie genre, date of release, etc.). One huge advantage to collaborative filtering (CF) is that it enables accurate recommendations to be made without prior knowledge of a given item [1, 2]. Collaborative filtering is a branch of algorithms that use item ratings from other users in conjunction with user history in order to make item recommendations for the target user [1]. In short, it only requires as input a matrix of item-user ratings. Two general classes of collaborative filtering are memory-based algorithms and model-based algorithms [3, 2].

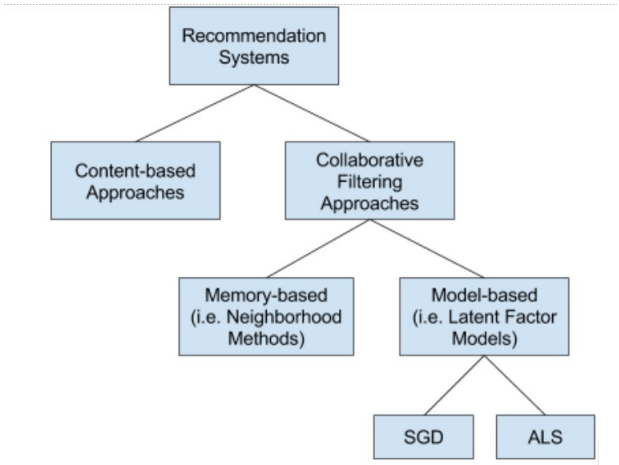

Figure 1: Visual Breakdown and Classification of Various Recommendation Systems.

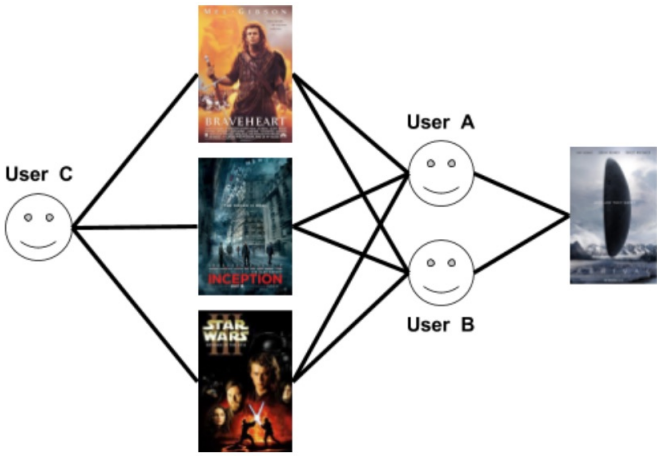

Figure 2: Memory-based Collaborative Filtering.

2.3 Memory-based vs. Model-based Collaborative Filtering

Memory-based algorithms, such as the neighborhood methods, use similarity between users to calculate preferences. For example, if users A, B and C all like the same three movies, then if users A and B both like a fourth movie, it is likely that user C will also like this movie. See Figure 2 for an illustration of this example. While these memory-based algorithms can typically be implemented quite easily and produce predictions with reasonable accuracy, their major drawback is scalability to real-world situations [2]. A cold-start problem is when a new user or item enters the system, and these memory-based algorithms do not address this issue [2]. This is where model-based algorithms have a great advantage over memory-based approaches. Many model-based algorithms employ matrix factorization to separate a given matrix into the product of two other matrices. One primary model-based approach is called the latent factor model, which aims to explain the ratings by characterizing both items and users on k factors (see Figure 3).

2.4 Latent Factor Model

Extending the underlying assumption of content filtering, we have the intuition that the rating of a user u of movie i may depend on several properties of the movie (eg, genre, leading actor, geared toward male/female, etc), where the rating may be affected by his/her orientation/preference toward a certain movie property. Conversely, from a movie's perspective, the rating that a given movie i got from user u may depend on the characteristic or demographic info of that person, where the rating may be affected by the attractiveness of this movie to people who possess certain characteristics.

To summarize, all of these properties and characteristics are factors that describe either a person or a movie and ultimately affects the ratings. More specifically, the ratings reflect how well the factors for a given person and movie align (i.e whether preference [person] vs. properties [movie], characteristics [person] vs. attractiveness [movie]). This forms the basic foundation of the model.

However, not all factors are created equal because each may have a varying degree of influence on the rating. An extreme example is that whether or not a person has gotten bit by a dog, may, in general, have little effect on the his/her ratings on movies, whereas cultural differences may heavily influence a person's preferences of movies. There are factors that carry much more influence on the rating than others, and therefore it's not hard to come up with the assumption that there are several "most important" factors (say we have k of them) that summarize the essence of this human-movie relationship. With just these factors, we can approximate the rating well enough.

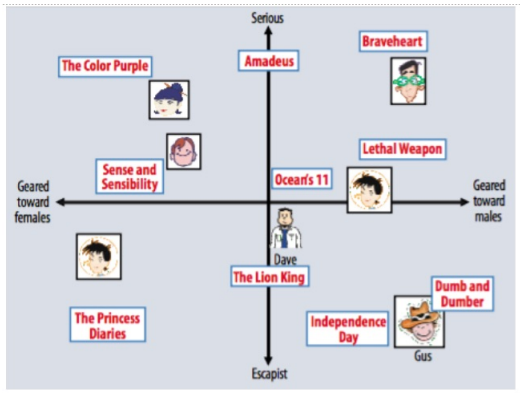

Figure 3: A Simplified Representation of the Latent Factor Model [4]

These k factors are are inferred from the given rating matrix R. Latent factor models use matrix factorization, in a similar fashion as the singular value decomposition (SVD), to yield an item-feature matrix Q and a user-feature matrix P [4]. Given a matrix R_{ui} of ratings for each user u to each item i, the users and items can be mapped to a joint latent factor space with dimensionality k [5]. More details regarding the actual factorization will be covered in the next section.

2.4.1 Matrix Factorization

To formulate the math, suppose the k latent factors of user u and movie i are known, with $q_i, p_u \in \mathbb{R}^k$ being the latent factor vectors for user u and movie i, respectively, as informally described in Section 2.4. The underlying assumption of latent factor models is that the true rating $\tilde{r}_{ui}$, can be approximated by a function of p_u and q_i.

$$\tilde{r}_{ui} \approx \hat{r} = f(p_u, q_i) \tag{1}$$

where $f(p_u, q_i)$ is some function describing how well p_u and q_i align. One intuitive choice of f is the inner product:

$$f(p_u, q_i) = p_u^T q_i \tag{2}$$

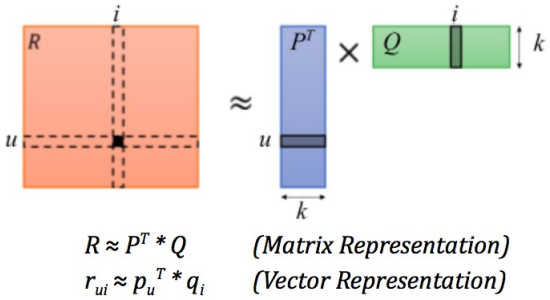

Figure 4: Matrix Factorization for the Latent Factor Model [6]

Then, with q_u, p_i given, it is easy to compute the estimated rating. Equation 2 gives rise to the Matrix factorization model, which, abstractly speaking, maps both users and items to a joint latent factor space of dimensionality k, such that user-item interactions are modeled as inner products in that space.

To get a better understanding of the matrix factorization models, let's take the matrix form of Equation 2:

$$\tilde{R} \approx \hat{R} = P^T Q \tag{3}$$

Where $\tilde{R}$ is the $m \times n$ matrix with $[\tilde{R}]_{ui} = \tilde{r}_{ui}$ that gives the true rating of user u on movie i, for all the u's and i's, P is the $k \times m$ matrix with uth column $P_{\cdot u} = p_u$, Q is the $k \times n$ matrix with ith column $Q_{\cdot i} = q_i$.

2.4.2 Relationship with SVD

The goal is then to find all of the p_u, q_i such that $\hat{r}_{ui} = q_u^T q_i$ best approximate $\tilde{r}_{ui}$. The problem can then be expressed as solving the following optimization problem:

$$\min_{P,Q} L(\tilde{R}, P, Q) \tag{4}$$

Where $L(\tilde{R}, P, Q)$ is the loss function evaluating the accuracy of the estimation, with smaller value of L indicating better accuracy. One popular choice of L is the root mean square error (RMSE) defined as the following:

$$RMSE(\tilde{R}, P, Q) = \frac{1}{mn} \sqrt{\sum_u \sum_i (\tilde{r}_{ui} - p_u^T q_i)^2} \tag{5}$$

And then the solution for following optimization problem will be the same as Equation 4.

$$\min_{P,Q} \sum_u \sum_i (\tilde{r}_{ui} - p_u^T q_i)^2 \tag{6}$$

Again taking the matrix form, notice that:

$$RMSE(\tilde{R}, P, Q) = \frac{1}{mn} \left\| \tilde{R} - P^T Q \right\|_F \tag{7}$$

Where $\|\cdot\|_F$ is the frobenius norm. And the problem becomes equivalent to:

$$\min_{P,Q} \left\| \tilde{R} - P^T Q \right\|_F \tag{8}$$

Now consider a problem that is closely related to Equation 8:

$$\min_{R=P^TQ} \left\| \tilde{R} - R \right\|_F \tag{9}$$

The goal is now to find $R = P^TQ$ best approximate $\tilde{R}$. From what we've learned in class, $\text{Rank}(P) \leq k$, $\text{Rank}(Q) \leq k$, which implies $\text{Rank}(R) \leq k$. And the problem can then be reformed to:

$$\min_{\text{Rank}(R)\leq k} \left\| \tilde{R} - R \right\|_F \tag{10}$$

The above expression, as you may recall from the lecture, can be solved by a low rank approximation with SVD.

2.4.3 Missing Ratings and Regularization

The above closed form solution may seem very appealing at fist glance. However, it suffers from a fundamental problem when applied to recommender systems. In the discussion in Section 2.4.2 the matrix $\tilde{R}$ used to compute the loss function is the true rating matrix that contains the ratings of each user u on each movie i for any u and i. Which means that we have to have all the rating. However, this will never be true for a recommender system (we won't need to predict any rating if we have them all). In fact, for all the recommender system the rating matix R in $\mathbb{R}^{m\times n}$ from the actual data would be quite sparse. Let κ be the set of indices (u, i) such that user u rated movie i in the given data. Let's define the sparsity of a rating matrix R:

$$\text{sparsity}(R) = \frac{mn - |\kappa|}{mn} \tag{11}$$

where $|\kappa|$ will be the number of ratings actually rated in R. To accommodate the model with sparsely filled R. The following loss function is proposed:

$$RMSE^{sparse}(R, P, Q) = \frac{1}{|\kappa|}\sqrt{\sum_{(u,i)\in\kappa}(r_{ui} - p_u^T q_i)^2} \tag{12}$$

where $r_{ui} = [R]_{ui}$. Equation 12 only accounts for the rating provided without penalizing entries that are not rated. With this model, there are $(m + n)k$ unknowns to be determined. On the other hand, with a sparse R, the known ratings set κ may have fewer elements than that. Therefore, to obtain a unique solution to the problem without over fitting the provided information with too much model parameters, a Tikhonov regularization term is appended to the loss function:

$$\min_{P,Q} \sum_{(u,i)\in\kappa} (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2) \tag{13}$$

This will be the primary model used for this lab. Thus, our focus shifts toward methods for minimizing this expression in order to learn P and Q so that predictions can be made. Two commonly used approaches for minimizing the previous equation are stochastic gradient descent (SGD) and alternating least squares (ALS). These are both useful for updating your model incrementally with each additional rating.

2.4.4 Stochastic Gradient Descent

One optimization technique of Equation 13 by Simon Funk [7] is a flavor of stochastic gradient descent. Considering the objective function of Equation 13, we have:

$$f(P, Q) = \sum_{(u,i)\in\kappa} (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2) \tag{14}$$

To do sampling, notice that Equation 14 can be decomposed to:

$$f(P,Q) = \sum_{(u,i)\in\kappa} \hat{f}_{ui}(P,Q) \tag{15}$$

where:

$$\hat{f}_{ui}(P,Q) = (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2) \tag{16}$$

on which we can further conduct batched coordinate descent with respect to p_u and q_i, respectively.

To implement SGD, the algorithm in this method loops through all the ratings in the training set, calculating the prediction error for each element with a rating. The prediction error for each given training case is calculated using Equation 17, which takes the value of the predicted rating (calculated from Equation 2), and subtracts it from the known true rating, r_{ui}. This error is used to update p_u and q_i (each vector at a time), using Equations 18 and 19 (for detailed derivation see Warm Up 1). A general procedure of the SGD algorithm as taken from Zhou is provided in Algorithm 1 [4].

while *not converged* **do**
  for $(u,i) \in \kappa$ **do**

$$e_{ui} \leftarrow r_{ui} - p_u^T q_i \tag{17}$$

$$p_u \leftarrow p_u + \gamma(e_{ui} q_i - \lambda p_u) \tag{18}$$

$$q_i \leftarrow q_i + \gamma(e_{ui} p_u - \lambda q_i) \tag{19}$$

  end
end

Algorithm 1: SGD

2.4.5 Alternating Least Squares

Another way to optimize Equation 13, proposed in [8] is alternating least square. Since p_u's and q_i's are both unknowns in the objective function, Equation 14 is non-convex, which makes it hard to solve.

However, it can be modified into a simpler problem if we fix either all of the p_u's or all of the q_i's. Considering all of the q_i's to be fixed, then the optimization problem becomes equivalent to:

$$\min_P \sum_{(u,i)\in\kappa} (r_{ui} - p_u^T q_i)^2 + \sum_u N_u \lambda \|p_u\|^2 \tag{20}$$

where N_u is the total number of movies user u has rated. More interestingly, the objective function in Equation 20 becomes separable:

$$f(P) = \sum_{(u,i)\in\kappa} (r_{ui} - p_u^T q_i)^2 + \sum_u N_u \lambda \|p_u\|^2 \tag{21}$$

$$= \sum_U \tilde{f}(p_U) \tag{22}$$

where:

$$\tilde{f}(p_U) = \sum_{(U,i)\in\kappa} (r_{Ui} - p_U^T q_i)^2 + N_U \lambda \|p_U\|^2 \tag{23}$$

This means that to solve the optimization problem in Equation 20, we only need to minimize $\tilde{f}(p_U)$ with respect to each p_U, which is simply a least squares problem that we have learned how to do in class.

This gives rise to the basic idea of ALS, which is alternating which one you fix and then performing least squares. While q_i's are fixed, the objective function can be optimized upon p_u's by solving a least square problem (Equation 24. For more details in solving the least square problem see Warm Up 2), and vise versa.

A general description of the ALS algorithm as taken from Zhou is provided in Algorithm 2 [3, 4].

Initialize Q with small random numbers;
Modify the first row of Q to average rating of the corresponding movie;
while *not converged* **do**
  for *each* u **do**

$$p_u \leftarrow (QD_uQ^T + N_u\lambda\mathbb{I})^{-1}QD_uR_{u\cdot}^T \tag{24}$$

  end
  for *each* i **do**

$$q_i \leftarrow (PD_iP^T + N_i\lambda\mathbb{I})^{-1}PD_iR_{\cdot i} \tag{25}$$

  end
end

Algorithm 2: ALS

where:

$$[D_u]_{ii} = \begin{cases} 1 & (u,i) \in \kappa \\ 0 & (u,i) \notin \kappa \end{cases} \tag{26}$$

are $n \times n$ diagonal matrices. Similarly,

$$[D_i]_{uu} = \begin{cases} 1 & (u,i) \in \kappa \\ 0 & (u,i) \notin \kappa \end{cases} \tag{27}$$

are $m \times m$ diagonal matrices. Also, $R_{\cdot i}$, $R_{u\cdot}$ are columns and rows of R, respectively. $\mathbb{I} \in \mathbb{R}^{k\times k}$ is the identity matrix.

2.4.6 Comparison of SGD and ALS

In general, it is easier to implement the SGD algorithm. For the ALS algorithm, it requires a linear solver to solve the least squares problem. In each step of ALS, a $k \times k$ linear system need to be solved which can be expensive depending on k. However, there are cases where ALS is preferable compared to SGD.

Consider, when the system is parallelizable (only possible for ALS), the system computes each q_i independently of the other item factors and computes each p_u independently of the other user factors. This gives rise to potentially massive parallelization of the algorithm[4].

Also, when the training set is not sparse, descent looping over each single training case—as gradient descent does—would not be practical. ALS can efficiently handle such cases.

3 Warm Up

1. **Gradient Derivation for SGD.** To gain appreciation for the mathematical basis of SGD in Section 2.4.4, derive the result represented in Equation 18 by analytically taking the gradient

of Equation 16 with respect to p_u.

2. **Derivation for ALS Solution.** To gain appreciation for the mathematical basis of ALS in Section 2.4.5, let's investigate the derivation for the ALS Equation 24 for updating p_u while fixing Q.

   (a) From the class, we know that a Tikhonov regularized least square problem can be represented in the form of
   $$\min_x \|Ax - b\|^2 + \mu \|x\|^2$$
   Reformulate Equation 23 to this matrix form. Find the corresponding A, b, μ, and x.

   (b) Now solve the regularized least square problem you derived in 2a

3. **Get familiarized with data set.** Load the MovieLens data set[1] and construct input ratings matrix R.

   (a) Download the available ***.zip file**[2] and extract the files into your current directory. Load the **u1.base** data into your workspace. Name it something useful as this will be your training set.

   (b) Take a look a the data contained in this matrix. Column 1 represents the userID, column 2 represents the movieID, and column 3 represents the rating for the corresponding user and movie. Note: you can discard column 4 as this is simply a timestamp.

   (c) Now, with this data, create a ratings matrix R as demonstrated in Figure 4. You should create your matrix with each row representing a different user's ratings for each movie and with each column representing a different movie's ratings from various users. (Hint: Because this will be a very sparse matrix, you can use the function **sparse** in MATLAB.)

4 Lab

1. **Getting start with the implementation.** Suppose we have the following training data matrix with columns 1, 2, and 3 representing the user ID, movie ID and rating. Also, suppose your P and Q are initialized as follows:

$$\text{train} = \begin{bmatrix} 1 & 2 & 1.6 \\ 1 & 4 & 2.6 \\ 1 & 6 & 2.4 \\ 1 & 8 & 0.6 \\ 2 & 4 & 2.8 \\ 2 & 3 & 2.0 \\ 2 & 7 & 0.5 \\ 3 & 2 & 3.0 \\ 3 & 3 & 3.8 \\ 3 & 1 & 4.0 \end{bmatrix} \quad P = \begin{bmatrix} 1.4 & 1.2 & 0.1 \\ 2.4 & 2.7 & 2.3 \end{bmatrix}$$

$$Q = \begin{bmatrix} 4.0 & 2.3 & 2.9 & 2.7 & 2.7 & 2.4 & 0.5 & 0.6 \\ 1.9 & 1.6 & 2.9 & 2.4 & 0.4 & 0.6 & 2.7 & 2.1 \end{bmatrix}$$

   (a) Implement a function SGDstep(train, P, Q, lambda, gamma) that performs one iteration of the SGD algorithm, where train is the training data set and inputs P, Q, gamma, and

[1] This data was generated by real users submitting their ratings on at least 20 movies. It has also been effectively split into 5-fold crossvalidation sets. In this lab, we used the first set, u1.base, and u1.test.

[2] Note: This dataset can be obtained from MovieLens website → "older datasets" → ml-100k.zip [9]

lambda are as described in Section 2.4.4. Set the return values for this function as the resulting P and Q from 1 iteration of SGD. (Hint: for better performance, you may want to iterate through the data in the original data structure directly as opposed to converting it to the rating matrix form)

(b) Perform 1 iteration of SGD on the training data given with $\lambda = 0.01$, $\gamma = 0.01$.

(c) Implement a function ALSstep(train, P, Q, lambda) that performs one iteration of the ALS algorithm. Again, train is the input training data set and inputs P, Q, and lambda are as described in Section 2.4.5. The return values will be the resulting P and Q after 1 iteration of ALS.

(d) Perform 1 iteration of ALS on the data given with $\lambda = 0.01$.

2. **Accuracy estimation.** Before you start running the decomposition algorithm on the dataset, it is probably better to test it on a smaller set. As mentioned above, when the rating matrix R is fully filled, low rank approximation with SVD is a closed form solution to the loss function minimization problem (which is basically minimizing the frobenius norm of the matrix when the matrix is fully populated).

   (a) Start with generating a 40 x 40 matrix fully filled with random numbers ranging from 0 to 5. Solve the rank 10 approximation problem with SVD. Compute the RMSE (sqrt(frobenius norm)) of the SVD estimation, this will be the "ground truth" of the real optimized decomposition error.

   (b) Complete the SGD algorithm and use it to approximate the same matrix. Solve the same optimization problem for only the loss function (in other words, set $\lambda = 0$). Run the algorithm with $\gamma = 0.01$ for 500 iterations. Compute the RMSE (sqrt(frobenius norm)) of the approximated matrix. (Hint: you can initialize your P and Q with random numbers to start with)

   (c) Similarly, complete the ALS algorithm to approximate the same matrix. Solve the same optimization problem for only the loss function (set $\lambda = 0$). Run the algorithm for 100 iterations. Compute the RMSE (sqrt(frobenius norm)) of the approximated matrix. (Hint: start with minimizing with regards to Q and initialize the first row of Q to be average rating of each movie)

   (d) What do you notice about the errors of these three different approximation?

3. **Running time estimation.** By now, you've essentially finished implementing the two algorithms and are certain that they work as expected, so let's do some analysis on the run time of the two programs. As mentioned in Section 2.4.3, the two algorithms scale up differently with respect to the sparsity.

   (a) Again, start with generating a 40 x 40 matrix fully filled with random numbers ranging from 0 to 5. This time, randomly select a subset of the matrix to create matrices with different sparsities (hint randperm may be a good way to select the entries from the matrix).

   (b) For each matrix generated, run SGD for 100 steps and ALS for 15 steps and record the run times for each algorithm. For this exercise, you can use $\lambda = 0.01$, $\gamma = 0.01$.

   (c) Plot the run time vs. sparsity for the 2 algorithms on the same plot. What can you conclude from the plot?

4. **Create a rating prediction system.** Now, you've got all the building blocks for a recommender system, so let's go ahead and apply it to our dataset.

(a) With **u1.base**, the training data you got from Warm Up 3 formulate a rank 20 matrix approximation problem, and solve it with our three different methods.

   i. Solve the problem with the SVD using rank 20 matrix approximation
   ii. Solve the problem with SGD for 100 iterations using $\lambda = 0.01$, $\gamma = 0.01$.
   iii. Solve the problem with ALS for 20 iterations using $\lambda = 0.01$.

(b) Now test your models from part 4a on the testing set, **u1.test**, and compute the RMSE errors of the three models. Compare the errors and explain the result using what have you learned from Section 2.4.2, 2.4.3.

(c) Repeat part 4a for SGD and ALS, but this time compute the RMSE on the testing and training datasets for each iteration. Then plot the RMSE vs. iteration curve.

(d) Set $\lambda = 0.00005$, repeat part 4c, and comment on your observations.

References

[1] Bugra Akyildiz. Alternating least squares method for collaborative filtering, 2014. Available at `https://bugra.github.io/work/notes/2014-04-19/alternating-least-squares-method-for-collaborative-filtering/`.

[2] Moritz Haller. Implementing your own recommender systems in python using stochastic gradient descent. Available at `http://online.cambridgecoding.com/notebooks/mhaller/implementing-your-own-recommender-systems-in-python-using-stochastic-gradient-descent-4#implementing-your-own-recommender-systems-in-python-using-stochastic-gradient-descent`.

[3] Moritz Haller. Predicting user preferences in python using alternating least squares. Available at `http://online.cambridgecoding.com/notebooks/mhaller/predicting-user-preferences-in-python-using-alternating-least-squares`.

[4] Yehuda Koren, Robert Bell, Chris Volinsky, et al. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009. Available at `https://datajobs.com/data-science-repo/Recommender-Systems-[Netflix].pdf`.

[5] Michael Hahsler. Recommender systems: From content to latent factor analysis, 2011. Available at `http://michael.hahsler.net/research/Recommender_SMU2011/slides/Recomm_2011.pdf`.

[6] Data Science Made Simpler. How do you build a 'people who bought this also bought that'-style recommendation engine, 2015. Available at `"https://datasciencemadesimpler.wordpress.com/tag/alternating-least-squares/`.

[7] Simon Funk. Netflix update: Try this at home, 2006. Available at `http://sifter.org/~simon/journal/20061211.html`.

[8] Agnes Jóhannsdóttir. Implementing your own recommender systems in python. Available at `http://online.cambridgecoding.com/notebooks/eWReNYcAfB/implementing-your-own-recommender-systems-in-python-2`.

[9] GroupLens. Movielens 100k dataset, 2016. Available at `http://grouplens.org/datasets/movielens/`.

A Solutions to Warm Up Exercises

Q1 Solution

With coordinate descent on one of the sample function

$$\hat{f}_{ui}(P,Q) = (r_{ui} - p_u^T q_i)^2 + \lambda(\|p_u\|^2 + \|q_i\|^2)$$

Compute the gradient with respect to p_u:

$$\nabla_{p_u} \hat{f}_{ui} = -2(r_{ui} - p_u^T q_i)q_i + 2\lambda p_u$$

with e_{ui} defined as

$$e_{ui} = r_{ui} - p_u^T q_i$$

The final update equation for p_u from sample function $\hat{f}_{ui}$ with learning rate γ will be

$$p_u \leftarrow p_u + \gamma(e_{ui}q_i - \lambda p_u)$$

Q2 Solution

(a)

The objective function

$$\begin{aligned}\tilde{f}(p_U) &= \sum_{(U,i)\in\kappa} (r_{Ui} - p_U^T q_i)^2 + N_U\lambda \|p_U\|^2 \\ &= \sum_{(U,i)} d_{Ui}(r_{Ui} - p_U^T q_i)^2 + N_U\lambda \|p_U\|^2\end{aligned}$$

where

$$d_{Ui} = \begin{cases} 1 & (U,i) \in \kappa \\ 0 & (U,i) \notin \kappa \end{cases}$$

Then,

$$\begin{aligned}\tilde{f}(p_U) &= \sum_{(U,i)} d_{Ui}(r_{Ui} - p_U^T q_i)^2 + N_U\lambda \|p_U\|^2 \\ &= \left\|D_U Q^T p_U - D_U R_{U.}^T\right\|^2 + N_U\lambda \|p_U\|^2\end{aligned}$$

Therefore,

$$\begin{aligned}A &= D_U Q^T \\ b &= D_U R_{U.}^T \\ \mu &= N_U\lambda \\ x &= q_U\end{aligned}$$

(b)

The solution of the regularized least square problem

$$\min_x \|Ax - b\|^2 + \mu \|x\|^2$$

is

$$x = (A^T A + \mu\mathbb{I})^{-1} A^T b$$

Plugging in the A, b, x obtained above, this results in the following:

$$\begin{aligned}q_U &= (QD_U^T D_U Q^T + N_U\lambda\mathbb{I})^{-1} QD_U^T D_U R_{U.}^T \\ &= (QD_U Q^T + N_U\lambda\mathbb{I})^{-1} QD_U R_{U.}^T\end{aligned}$$

Q3 Solution

For all parts, see Figure 5

Warm Up Section - Q3

```
clc; close all; clear;

% Load Data Set
%   Note: The exact syntax depends where your data set is located
train = importdata('../u1.base');

m = 943;                    % number of userIDs
n = 1682;                   % number of movieIDs
i = train(:,1);             % userIDs
j = train(:,2);             % movieIDs
s = train(:,3);             % ratings
R = sparse(i,j,s,m,n);      % construct sparse R matrix
```

Figure 5: Q3 Solution

B Solutions to Lab Exercises

Q1 Solution

(a) See Figure 6

SGD Step Function

```
function [P,Q] = SGDstep(train,P,Q,lambda,gamma)
for idx = 1:size(train,1)
    u = train(idx,1);
    i = train(idx,2);
    e = train(idx,3) - P(:,u)'*Q(:,i);
    P(:,u) = P(:,u)+gamma * ( e * Q(:,i) - lambda * P(:,u));
    Q(:,i) = Q(:,i)+gamma * ( e * P(:,u) - lambda * Q(:,i));
end
end
```

Figure 6: Q1 Solution - SGD Step Function

(b) See Figure 7 run SGDstep section

(c) See Figure 8

(d) See Figure 7 run ALSstep section

Q2 Solution

(a) See Figure 9 SVD section

(b) See Figure 9 SGD section

(c) See Figure 9 ALS section

(d) From the error calculated, it can be seen that all three algorithms give roughly the same RMSE (error) for low rank approximation of a fully filled matrix.

Q3 Solution

(a) See Figure 10 Part(a)

(b) See Figure 10 Part (b)

(c) As can be seen Figure 10 Part (c), as the matrix get more sparse, the SGD run time gets faster. However, the performance of ALS is relatively unaffected. Thus, if the rating matrix is more dense, ALS should be use to solve the optimization problem.

Q4 Solution

(a) 4(a)i See Figure 11; 4(a)ii 4(a)iii See Figure 12

(b) The error for the testing data with SVD, SGD, ALS are $2.89, 0.982, 0.940$, respectively. It can be seen that the error with SVD is a lot larger compared to SGD and ALS. This is because the rating data is not fully filled and thus SVD is actually trying to fit to data that is nonexistent.

(c) For the code portion, see Figure 12. For plots see Figure 15

(d) See Figure 15 This plot shows the effect of regularization. As can be observed in 15, when the regularization factor is too small, the algorithms are prone to overfitting, i.e. as number of iterations goes up, the error on the training set decreases while error on testing set increases.

Lab Section - Q1 Main Functionality Code

```
clear; clc;
```

Q1 - setup

```
train = [1    2    1.6
         1    4    2.6
         1    6    2.4
         1    8    0.6
         2    4    2.8
         2    3    2.0
         2    7    0.5
         3    2    3.0
         3    3    3.8
         3    1    4.0];
P = [1.4    1.2    0.1; 2.4    2.7    2.3];
Q =  [4.0,2.3,2.9,2.7,2.7,2.4,0.5,0.6; 1.9,1.6,2.9,2.4,0.4,0.6,2.7,2.1];
m = size(P,2);
n = size(Q,2);
k = size(P,1);
lambda = 0.01;
gamma = 0.01;
R = sparse(train(:,1),train(:,2), train(:,3),m,n);
```

Q1 - run SGD step function

```
[P,Q] = SGDstep(train,P,Q,lambda,gamma)
```

```
P =

    1.0371    0.7558    0.0010
    2.0535    2.1442    2.2138


Q =

  Columns 1 through 7

    3.9996    2.2294    2.8331    2.5625    2.7000    2.3834    0.4536
    1.8940    1.4594    2.6469    2.0966    0.4000    0.5669    2.5683

  Column 8

    0.5527
    2.0063
```

Q1 - run ALS step function

```
[P,Q] = ALSstep(R,P,Q,lambda)
```

```
P =

    0.9152    0.8656    0.6773
    0.0187    0.0166    0.7527


Q =

  Columns 1 through 7

    2.6168    1.7100    2.2645    2.9882         0    2.5903    0.5698
    2.9083    2.3632    2.9080    0.0490         0    0.0529    0.0109

  Column 8

    0.6476
    0.0132
```

Figure 7: Q1 Solution - Main Functionality

ALS Step Function

```
function [P,Q] = ALSstep(train,P,Q,lambda)
R_index = train ~= 0;
k = size(P,1);
E = eye(k,k);
for i = 1:size(train, 1)
    % get n_pi --> number of items user i has rated
    n_pi = sum(R_index(i,:));
    if (n_pi == 0)
        n_pi = 1;
    end

    % Least squares solution
    Au = Q*(diag(R_index(i,:)) * Q') + lambda * n_pi * E;
    Vu = Q*(diag(R_index(i,:)) * train(i,:)');
    P(:,i) = inv(Au) * Vu;
end

% Fix P and estimate Q
for  i = 1:size(train, 2)
    % get n_qi
    n_qi = sum(R_index(:,i));
    if (n_qi == 0)
        n_qi = 1;
    end

    % Least squares solution
    Ai = P*(diag(R_index(:,i)) * P') + lambda * n_qi * E;
    Vi = P*(diag(R_index(:,i)) * train(:,i));

    Q(:,i) = inv(Ai) * Vi;
    %         toc
end
end
```

Figure 8: Q1 Solution - ALS Step Function

Lab Section - Q2 Main Functionality Code

setup;

```
clear
clc
rng(0);
m = 40;
n = 40;
R = rand(m,n)*5;
k = 10;

[I,U] = meshgrid(1:n,1:m);
U = reshape(U,m*n,1);
I = reshape(I,m*n,1);
train = reshape(R,m*n,1);
train = [U,I,train];
test = train;
```

Q2 - SVD

```
[u_svd,s_svd,v_svd] = svd(R);
A_SVD = u_svd(:,1:k)*s_svd(1:k,1:k)*v_svd(:,1:k)';
```

Q2 - SGD

```
[P,Q,stat] = SGD(train, test, m,n, k,0,500);
fprintf('SGD: Elapsed time is %d seconds.\n', stat.time);
A_SGD = P'*Q;
```

```
SGD: Elapsed time is 4.287569e+00 seconds.
```

Q2 - ALS

```
[P,Q,stat] = ALS(train, test, m,n, k,0,100);
fprintf('ALS: Elapsed time is %d seconds.\n', stat.time);
A_ALS = P'*Q;
```

```
ALS: Elapsed time is 7.581333e-01 seconds.
```

Q2 - accuracy analysis

```
e_SVD = sqrt(mean(mean((A_SVD-R).^2)));
e_SGD = sqrt(mean(mean((A_SGD-R).^2)));
e_ALS = sqrt(mean(mean((A_ALS-R).^2)));

fprintf('RMSE of SVD is %d\n', e_SVD);
fprintf('RMSE of SGD is %d\n', e_SGD);
fprintf('RMSE of ALS is %d\n', e_ALS);

diff_V_G = sqrt(mean(mean((A_SGD-A_SVD).^2)))
diff_V_A = sqrt(mean(mean((A_ALS-A_SVD).^2)))
```

```
RMSE of SVD is 8.700035e-01
RMSE of SGD is 1.764549e+00
RMSE of ALS is 8.700035e-01

diff_V_G =

    1.6063


diff_V_A =

   2.4781e-05
```

Figure 9: Q2 Solution - Main Functionality

Lab Section - Q3 Main Functionality Code

```
clear
clc
```

Part (a)

```
rng(0);
m = 40;
n = 40;
R = rand(m,n)*5;
k = 10;

[I,U] = meshgrid(1:n,1:m);
U = reshape(U,m*n,1);
I = reshape(I,m*n,1);
A = reshape(R,m*n,1);
A = [U,I,A];
% test = train;

N = 10;
ALST = zeros(1,N);
SGDT = zeros(1,N);
sparsity = linspace(0.01, 1,N);
```

Part (b)

```
tic
for i = 1:N
        idx = randperm(m*n,ceil(n*m*(1-sparsity(i))));
        train = A(idx,:);
        test = train;

        [~,~,stat] = SGD(train, test,m,n, k,0.01,100);
        SGDT(i) = SGDT(i)+stat.time;
        [~,~,stat] = ALS(train, test,m,n, k,0.01,15);
        ALST(i) = ALST(i)+stat.time;
end
toc
```

```
Elapsed time is 0.117071 seconds.
```

Part (c)

```
   figure
plot(sparsity,ALST);
hold on
plot(sparsity,SGDT); title('Running Time vs. Sparsity');
xlabel('Sparsity'); ylabel('Time'); legend('ALS','SGD');
```

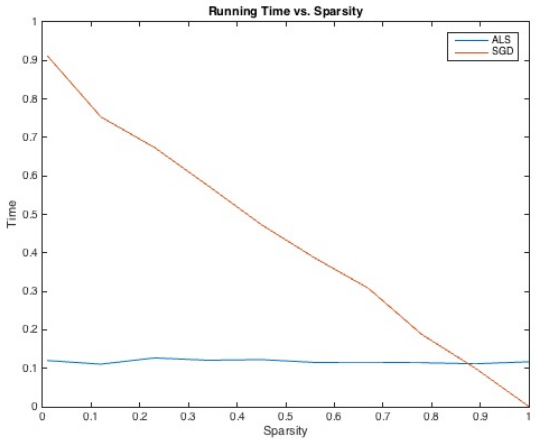

Figure 10: Q3 Solution - Main Functionality

Lab Section - Q4 Main Functionality Code

```
clc; close all; clear;
% Load Data Set
%   Note: The exact syntax depends where your data set is located
train = importdata('../u1.base');
test = importdata('../u1.test');
% Actual Matrix Factorization
k = 20;
m = 943;
n = 1682;
```

SVD

```
disp('-- Q4: SVD Approximation --');
tic
i = train(:,1);
j = train(:,2);
s = train(:,3);
R = sparse(i,j,s,m,n);
[U,S,V] = svds(R,k);
P = (U*S)';
Q = V';
errorTrain = RMSE(train,P,Q)
errorTest = RMSE(test,P,Q)
toc
```

```
-- Q4: SVD Approximation --

errorTrain =

    2.3014


errorTest =

    2.8880

Elapsed time is 3.239342 seconds.
```

Figure 11: Q4 Solution - Main Functionality - part 1

SGD

```
disp('-- Q4: SGD Approximation --');
tic
[P,Q,stat] = SGD(train, test,m,n, k,lambda, 'debugMode', true);
errorTrain = RMSE(train,P,Q)
errorTest = RMSE(test,P,Q)
rmseTrain = stat.eTrain;
rmseTest = stat.eTest;
figure
subplot(2,2,1)
plot(rmseTrain);
hold on
plot(rmseTest); title('SGD Approx: RMSE vs. Iterations');
xlabel('# of Iterations'); ylabel('RMSE');
legend('Training Data','Testing Data'); axis([0 100 0.5 4]);
toc
```

Figure 12: Q4 Solution - Sample Solution for SGD. ALS can be solved using the same code framework, but using the ALS function instead of the SGD function. Additionally, the axis will change for ALS. Then, each of these is repeated for $\lambda = 5 \times 10^{-5}$.

SGD Completed Function

```
function [P,Q,stat] = SGD(train, test, m, n, k, varargin)
p = inputParser;                          % parse the input arguments
addOptional(p,'lambda',0.1);
addOptional(p,'maxIt',100);
addOptional(p, 'gamma', 0.01);
addParameter(p,'debugMode', false);

parse(p,varargin{:});
debugM = p.Results.debugMode;

max_iter = p.Results.maxIt;               % setup SGD
gamma = p.Results.gamma;
lambda = p.Results.lambda;

t = 0;                                    % initialize paramters

rng(0);                                   % for reproducibility
P = 3* rand(k,m);
Q = 3* rand(k,n);

if debugM
    rmse = zeros(max_iter, 2);
end
tic
while t <= max_iter

    [P,Q] = SGDstep(train,P,Q,lambda,gamma);

     if debugM
        rmse(t+1,2) = RMSE(test,P,Q);  % compute the root mean square error
        rmse(t+1,1) = RMSE(train,P,Q);
     end
     t = t+1;
end
runT = toc;
if debugM
    stat.eTrain = rmse(:,1)';
    stat.eTest = rmse(:,2)';
end
stat.time = runT;
stat.lambda = lambda;
stat.iter = t;
end
```

Figure 13: Q2 Solution - SGD Complete Function

ALS Complete Function

```
function [P,Q,stat] = ALS(train, test, m, n, k, varargin)
p = inputParser;                        % parse the input arguments
addOptional(p,'lambda',0.1);
addOptional(p,'maxIt',20);
addParameter(p,'debugMode', false);

parse(p,varargin{:});
debugM = p.Results.debugMode;

max_iter = p.Results.maxIt;             % setup ALS
lambda = p.Results.lambda;

i = train(:,1);                         % UserID index
j = train(:,2);                         % MovieID index
s = train(:,3);
R = sparse(i,j,s,m,n);
R_index = R ~= 0;

t = 0;                                  % initialize paramters

rng(0);                                 % for reproducibility
P = 3* rand(k,m);
Q = 3* rand(k,n);
% Assigns first row of Q to be average rating for that movie
for i = 1:n
    if sum(R_index(:,i)) > 0
        Q(1,i) = sum(R(:,i)) / sum(R_index(:,i));
    end
end

if debugM
    rmse = zeros(max_iter, 2);
end
tic

while t <= max_iter

    [P,Q] = ALSstep(R,P,Q,lambda);

     if debugM
        % compute the root mean square error
        rmse(t+1,2) = RMSE(test,P,Q);
        rmse(t+1,1) = RMSE(train,P,Q);
     end
      t = t+1;
end
runT = toc;
if debugM
     stat.eTrain = rmse(:,1)';
    stat.eTest = rmse(:,2)';
end
stat.time = runT;
stat.lambda = lambda;
stat.iter = t;
end
```

Figure 14: Q2 Solution - ALS Complete Function

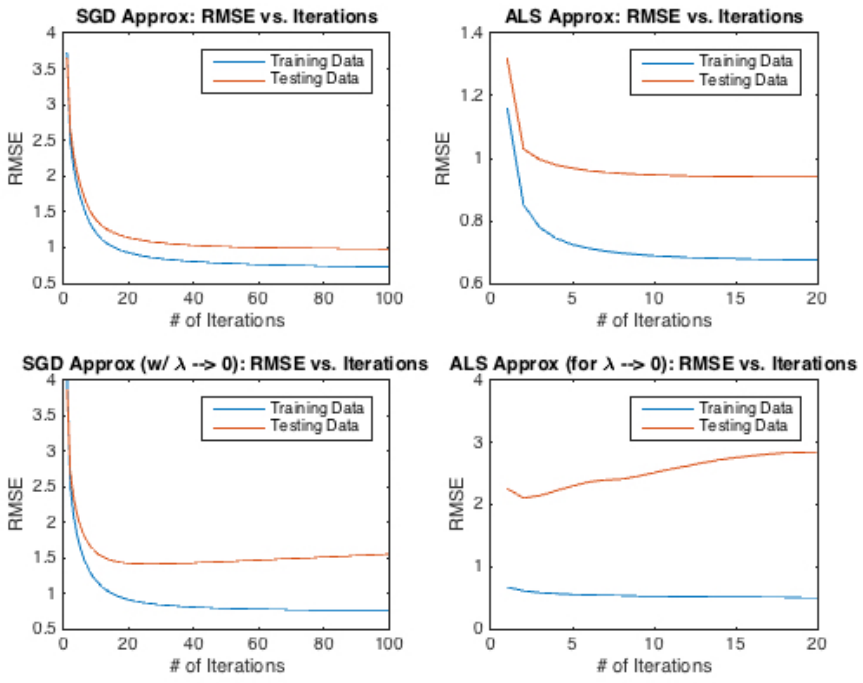

Figure 15: Q4 Solution - Number of iterations vs. RMSE